Waku Message UID (v2)

Created time	@February 1, 2023 4:41 PM
Last edited time	@February 2, 2023 11:16 AM
Scope	Product
Tags	adr
Issue link	
Related pages	

Context and previous attempts

Message deduplication in Waku Relay (Gossipsub)

The *Message Cache* and the *Seen Cache* perform m*essage* deduplication i*n* W*a*ku Relay*.* These two deduplication structures rely on a unique ID which, at the moment of writing, is computed as follows:

```
message_id: [u8; 32] = sha256(WakuMessageBuffer)
```

These structures provide a limited-time message deduplication capability o keep the memory footprint low. The *Message Cache* provides a short-term deduplication capability (~5 heartbeat periods). The *Seen Cache,* implemented as a bloom filter, provides a longer-term deduplication capability (~2 minutes).

Based on the fact that `WakuMessage` is formatted using protocol buffers, and this serialization mechanism is not deterministic, there is a possibility that the *Seen Cache* fails to lead to duplicated messages by just reordering the different fields that are serialized.

Libp2p's Gossipsub message validation

The Gossipsub v1.0 specification states the following about the message validation:

> Payload processing will validate the message according to application-defined rules and check the `seen` cache to determine if the message has been processed previously.

Essentially, the specification leaves the message validation to the application.

Consequently, the principal Gossipsub implementations (Nim, Go and Rust) do not implement message integrity or ID validation. And by extension, this is also true for the Waku Relay protocol, as it is an opinionated version of the Gossipsub protocol.

Message delivery order guarantees in Waku Relay

Gossipsub protocol, due to its flood-distribution nature, cannot guarantee that all messages will follow the same distribution path. So there is no ordering guarantee per se.

But if two messages are published separately enough in time, we can assume that both messages will be delivered in a predictable order. The maximum latency of the network determines this minimum separation.

Records deduplication in Waku Archive

At the moment of writing, the Waku Archive SQLite backend implementation considers two messages equal if both messages share the same `pubsub_topic`, `content_topic`, `timestamp` and `payload`. This criterion is baked into the SQLite's table `PRIMARY KEY`. Messages that do not match the criteria are treated as duplicates.

At the moment of querying via the Waku Store protocol, the results are ordered by the columns `stored_at` (equal to the message timestamp if present or to message arrival time otherwise) and `id` (computed as `Sha256(content_topic, payload)`).

Records retrieval via Waku Store

At the moment of writing, the *Waku Store* protocol provides an API supporting retrieving a filtered list of messages from a remote node. Additionally, as the maximum size supported for a message is 1MB, the *Waku Store* query protocol supports a pagination mechanism. This helps reduce the amount of data downloaded per query to a maximum of 100 messages (100 MB).

However, there is a limitation when verifying missing messages in a node's history. Downloading large amounts of data, in the order of hundreds of megabytes or even gigabytes, is not a time and bandwidth-efficient method for this task.

Goal

Definition of a Waku message uniqueness identifier that can be used to deduplicate Waku messages across the solution.

No goal

- Build a blockchain. Consistency guarantees between archive-capable nodes are out of the scope of the present document.
- Address or mitigate all vulnerabilities to which Waku Relay (and Gossipsub) are susceptible.

Use cases

- Message deduplication in the network.
- Message deduplication in the Waku Archive backend (e.g., in a shared backend setup).
- Bandwidth-efficient Waku Archive synchronization.

Requirements

- Length limited.
- Low to negligible collision probability.
- Not leaking information (e.g., sender's key).
- Application specific (e.g., lexicographic sortable).
- Uniqueness is global to the network.
  - Global to all the nodes publishing in a certain network (pub-sub topic).
  - Global to the Gossipsub's pub-sub topics certain store node is subscribed to.
- Act as a message integrity check.
- As an open network, all nodes should be able to perform a message ID validation.

Pre-requites

The Waku Message's `meta` attribute

The Waku Message's `meta` attribute is an arbitrary application-specific variable-length byte array with a maximum length limit of 32 bytes (2^256 possibilities).

The message's `meta` field MUST be present and have a length greater than zero in the non-ephemeral messages (those persisted by the Waku Archive durability service).

The Waku Message UID (MUID)

The Waku Message UID is a two-part variable length identifier that can unequivocally identify and deduplicate the messages in a Waku network.

The MUID comprises two parts: *message checksum* and *application-specific variable-length metadata.*

```
muid: [u8; 64] = concat(checksum, metadata)
```

The maximum length for the MUID is 64 bytes.

The *checksum* part

It is a computable 32 bytes fixed-length checksum based on the content of the Waku Message. It is defined as follows:

```
checksum: [u8; 32] = sha256(network_topic, WakuMessage.topic, WakuMessage.meta, WakuMessage.payload)
```

The *checksum* part ensures the integrity of the Waku Message contained in the Gossipsub payload. As any node in the network can compute it, the message integrity can be verified.

The *metadata* part

It is an application-specific part extracted from the Waku Message's `meta` attribute.

Message uniqueness considerations

Two messages are considered equal if they have the same `network topic`, `content topic`, `meta` attribute, and `payload`. As the MUID is derived from the message content, both messages share the same.

The application should provide different `meta` attributes to different messages to avoid collisions in the relay and archive collisions.

Example message metadata schemas

Applications should specify the schema for the Waku Message's `meta` attribute. The selected schema will affect the application's messages' privacy, security and message collision probability.

These are some example schemas that could be used:

- **Timestamp (e.g., int64 Unix Epoch nanoseconds timestamp):**
  - **PRO:** Simple, performant generation, "backwards compatible", fine grain sortable at archive query results (precision in ns).
  - **CON:** Prone to collision/message duplication, traceable/graph learning
- **ULID:**
  - **PRO:** Medium complexity, performant generation, negligible collision probability, fine grain sortable at archive query results (precision in ms).
  - **CON:** traceable/graph learning
- **UUID (e.g., UUID v4):**
  - **PRO:** Medium complexity, performant generation, negligible collision probability, no traceability (random data).

    - **CON:** Coarse sortability at archive query time.
- **Noise Sessions (e.g., encrypted metadata/info):**
    - **PRO:** Negligible collision probability (if the content is well thought), contains metadata, non-traceable (looks like random data).
    - **CON:** High complexity, not-so-performant generation (hashing, encryption), not sortable at archive query time.

Waku Relay: deduplication and integrity

Based on the low collision probability of some of the schemas described above, this MUID could be used as the message and seen caches key.

A message that reuses the same ID with a different payload within the *Message Cache* window won't be relayed. In the same way, if it is replayed within the *Seen Cache* window, it won't be received by subscribers.

Additionally, as all nodes can compute the *checksum* part of a message ID, a validator can be integrated to guarantee the Waku Message integrity.

Waku Archive and Waku Store: durable streams

> ℹ️ The terms *event*, *message*, and *record* are synonyms in the *Waku Archive* context. Thus they can be used interchangeably. Although the *record* term is preferred over the others when talking about individual items in an events log.

From the Waku product document's terms and concepts:

> Waku platform **events** are organized and durably stored in **topics,** also called **content topics.** And a sequence of events published on the same **topic** constitutes an **event stream**.

Event recording

The *Waku Archive* stream durability service is responsible for recording and persisting in a long-term storage system the events that occurred in a certain event stream. This persistence follows two rules:

- Messages should be stored following the arrival order (FIFO).
- Messages should be deduplicated based on the MUID (the same criteria used in the *Waku Relay* layer).

Optionally, messages can be tagged with arrival time timestamps for coarse-ordering purposes.

Queriable event log

Additionally to the event recording system, the *Waku Archive* service features a searchable log functionality and provides an interface for retrieving messages from the previously described event recording system. And the *Waku Store* historical messages query protocol sits on top of this interface, making it easily accessible through a remote procedure call (RPC) interface.

Waku Store's bandwidth-consumption optimization

Based on the assumption that MUIDs are globally unique to all messages. We can understand archive-capable nodes as key-value stores. The MUID would be the key, and the message would be the value.

With that approach, the current *Waku Store* protocol RPCs can be extended to support a message ID query mechanism. The new Waku Store protocol APIs will look like this:

- Query a list of messages based on certain filter criteria (e.g., network, content topic, time range, etc.).
- Query a list of MUIDs based on certain filter criteria (e.g., network, content topic, time range, etc.).
- Query messages by a MUIDs list.

With a maximum size of 32 bytes, the number of UIDs per query response can be higher, saving bandwidth and reducing the "time-to-sync" metric.

It is discretionary to the application in which APIs are used and when. For example, bootstrapping the node's history and getting the first 50 messages that fit the screen for a hypothetical messaging app can make a difference in the perceived UX. Once bootstrapped the node's message history, the same application could use the MUID-based query mechanism to efficiently retrieve the missing messages and complete the rest of the messages' history.

Waku Store's message decryption optimization

Additionally, as the MUID must contain application-specific metadata, a Waku Store client can identify the messages in the list of MUIDs retrieved from the Waku Archive.

This will reduce bandwidth consumption and the CPU load derived from downloading a big list of messages from another's node history and decrypting all the messages to filter only the application ones.

Waku Store's message integrity check

In the same way, the Waku Relay nodes can validate a message by computing the checksum part of the message; a Waku Store client can determine if any record has been maliciously modified and certify the integrity of the received entry.

Conclusions and future work

This proposal extends the current model and tries to unify the Waku platform's relay (*Waku Relay)* layer with the platform's stream durability functionality (*Waku Archive*) and the stream history query functionality (*Waku Store)*.

Eventually, this UID-based history synchronization mechanism has the potential to be evolved into a fully-fledged history synchronization mechanism. Due to this unified approach, it has the potential to be added as a Gossipsub extension adding a "durable stream capability" to the protocol.

An in-depth privacy and security analysis is pending.